TU Clausthal

Clausthal University of Technology

LTL Model Checking with Logic Based Petri Nets

Tristan M. Behrens and Jürgen Dix

IfI Technical Report Series IfI-07-04

Department of Informatics
Clausthal University of Technology

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Wojciech Jamroga
Contact: wjamroga@in.tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

LTL Model Checking with Logic Based Petri Nets

Tristan M. Behrens and Jürgen Dix

Department of Informatics, Clausthal University of Technology
Julius-Albert-Straße 4, 38678 Clausthal, Germany
{behrens,dix}@in.tu-clausthal.de

Abstract

In this paper we consider *unbounded model checking* for systems that can be specified in *Linear Time Logic*. More precisely, we consider the model checking problem "$\mathcal{N} \models \alpha$", where $\mathcal{N}$ is a generalized Petri net (SLPN) (which we have introduced in previous work), and α is an LTL formula. We solve this problem by using results about the equivalence of LTL formulae and Büchi automata.

1 Introduction

Model checking [2] is the problem to decide $M \models \alpha$: Is the system M a model of a logical formula α or not? Model checking together with *theorem proving* are two major techniques for *design validation at compile time*. Whereas systems based on theorem provers often rely heavily on user interaction, model checkers are fully automatic and deliver in addition a counter-example if an invalid execution or state of a system is detected.

Systems M are usually represented as *Kripke structures*. When dealing with concurrent systems formulæ are often expressed in *Computational Tree Logic* (CTL) or *Linear Time Logic* (LTL) [4]. Established model checkers rely on different theoretical foundations. For example *automata theory* has been investigated in the past and the model checking problem has been reduced to certain algorithms from graph theory.

Concurrent systems M can be described very nicely with *Petri nets*. Petri nets are mathematical structures based on simple syntax and semantics. They have been applied to model and visualize *parallelism*, *concurrency*, *synchronization* and *resource-sharing*. Different Petri net formalisms share several basic principles so that many theoretical results can be transferred between them [3]. In previous work [1] the authors have introduced a new type of Petri nets, *Simple Logic Petri Nets* (*SLPN*), to model concurrent systems that are *based on logical atoms* (e.g. multiagent systems specified in AgentSpeak).

In this paper we firstly recall the classical results about model checking and Büchi automata (Section 2). Our own work starts with Section 3, where we recall the class *SLPN* of *Simple Logic Petri nets* introduced in [1]. Then we show how to construct a

Büchi automaton from an *SLPN* (Section 4), via a Kripke structure. In Section 5 we use the results from Sections 2–4 to solve the model checking problem. Finally, Section 6 discusses related and future work. We conclude with Section 7.

2 LTL and Büchi Automata

Linear time logic (LTL) is a propositional logic with an additional operator for time. There is a very useful equivalence between LTL formula and Büchi automata—automata that operate on infinite words—a relationship that is crucial for model checking.

DEFINITION 1 (LTL Syntax [4]). *Let AP be a set of* atomic propositions*. The language LTL_{AP} is constructed as follows:*

- $AP \subset LTL_{AP}$ *and* $\top, \bot \in LTL_{AP}$*,*
- *if* $\alpha \in LTL_{AP}$ *then* $\neg\alpha \in LTL_{AP}$*,*
- *if* $\alpha, \beta \in LTL_{AP}$ *then* $\alpha \wedge \beta \in LTL_{AP}$ *and* $\alpha \vee \beta \in LTL_{AP}$*,*
- *if* $\alpha, \beta \in LTL_{AP}$ *then* $\bigcirc\alpha \in LTL_{AP}$*,* $[\alpha U \beta] \in LTL_{AP}$ *and* $[\alpha R \beta] \in LTL_{AP}$*.*

DEFINITION 2 (Semantics of LTL). *Let $\pi : s_0, s_1, s_2, \ldots$ be an (infinite) sequence of states of a system, generally π_i shall represent the sequence $s_i, s_{i+1}, s_{i+2}, \ldots$. Furthermore let $p\alpha$ be an atomic proposition and α and β LTL-formulæ. Then:*

- $\pi \models ap$ *iff* ap *is true in* s_0 *and* $\pi \models \neg\alpha$ *iff not* $\pi \models \alpha$
- $\pi \models \alpha \wedge \beta$ *iff* $\pi \models \alpha$ *and* $\pi \models \beta$*;* $\pi \models \alpha \vee \beta$ *iff* $\pi \models \alpha$ *or* $\pi \models \beta$
- $\pi \models \bigcirc\alpha$ *iff* $\pi_1 \models \alpha$
- $\pi \models [\alpha U \beta]$ *iff* $\exists i \geq 0 : \ \pi_i \models \beta$ *and* $\forall j, 0 \leq j \leq i : \ \pi_j \models \alpha$
- $\pi \models [\alpha R \beta]$ *iff* $\pi \models \neg[\neg\alpha U \neg\beta]$

The unary temporal operator $\bigcirc\alpha$ denotes that the formula α is true in the *next* state, whereas the binary temporal operator $\alpha U \beta$ (*until*) denotes that the formula α holds *until* β becomes true and $\alpha R \beta$ denotes that a α holds until it is *released* by β. We can express other modalities as follows: $\Diamond\alpha$ (*finally*, a formula will be true in the future) is equivalent to $[\top U \alpha]$ and $\Box\alpha$ (*globally*, a formula is true in all time moments from now on) is equivalent to $[\bot U \alpha]$.

Büchi automata are finite automata that read infinite words. Infinite words are suitable when dealing with nonterminating systems (operating systems, agents).

DEFINITION 3 (Büchi Automaton [9]). *The tuple $\mathcal{B} = \{\Sigma, Q, \Delta, Q_0, A\}$ with*

Σ *a finite alphabet,*

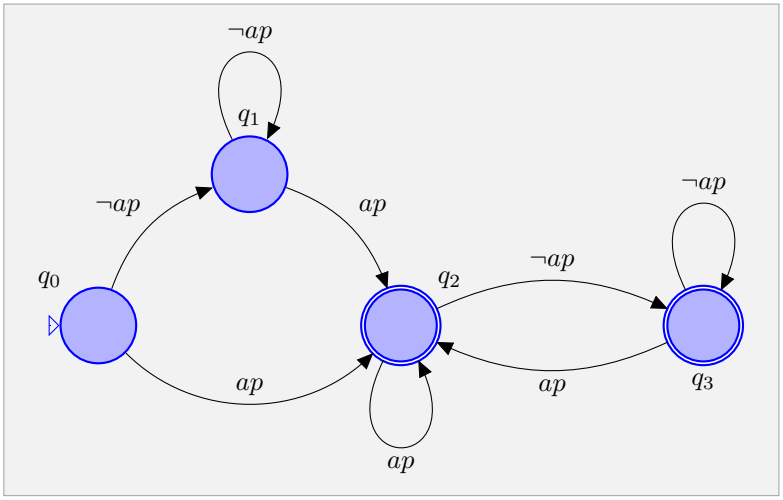

Figure 1: The Büchi automaton $\mathcal{B}_{\Diamond ap}$ derived from the formula $\Diamond ap$.

Q a finite set of states,

$\Delta \subseteq Q \times \Sigma \times Q$ a transition relation,

$Q_0 \subseteq Q$ a set of initial states,

$A \subseteq Q$ a set of accepting states,

is called Büchi automaton. *It accepts an infinite word w if in the course of parsing it at least one accepting state is visited* infinitely often.

Büchi automata are well suited for LTL:

THEOREM 1 (Gerth et al. ([5])). *For each LTL formula α there exists a Büchi automaton B_α that generates exactly those infinite sequences Π of states for which α holds.*

The proof is by Gerth's algorithm [5] which takes an LTL formula in positive normal form and generates an equivalent Büchi automaton. This is but one of many algorithms for deriving Büchi automata from LTL-formulæ [10]. It has two main advantages: the automaton can be generated simultaneously with the generation of a model and the algorithm proved to have a good average case complexity.

EXAMPLE 1 (A Simple Büchi Automaton).
Figure 1 shows the Büchi automaton generated from the LTL formula $\Diamond ap \equiv [\top U ap]$.

3 Simple Logic Petri Nets

In this section we introduce the class *SLPN* of Simple Logic Petri Nets. Let VAR be a finite set of *variables*, $CONST$ be a finite set of *constants*, $FUNC$ be a finite set of *function-symbols*, $TERM$ be the (infinite) set of *terms* constructed from terms, constants and atoms. Let $PRED$ be a set of *predicate symbols*, A^+ be the set of *positive atoms* and A^- be the set of *negative atoms* constructed using predicates and terms. Let $L = A^+ \cup A^-$ be the set of literals, X_{grnd} be the set of all *ground atoms* in each subset $X \subseteq L$. Finally, let $var\text{-}of : 2^L \to 2^{VAR}$ be the function that selects all variables that are contained in a given set of literals.

Petri nets are often referred to as *token games* and are usually defined by first providing the topology and then the semantics. The topology is a net structure, i.e. a bipartite digraph consisting of *places* and *transitions*. Defining the semantics means (1) defining what a state is, (2) defining when a transition is enabled and, finally, (3) to define how tokens are consumed/created/moved in the net. We refer to the notion that tokens are consumed and created by *firing transitions*—moving is made possible by *consuming* and *creating*. Our most basic definition is as follows:

DEFINITION 4 (Simple Logic Petri Net)**.** *The tuple $\mathcal{N} = \langle P, T, F, C \rangle$ with*

$P = \{p_1, \ldots, p_m\}$ *the set of places,*

$T = \{t_1, \ldots, t_n\}$ *the set of transitions,*

$F \subseteq (P \times T) \cup (T \times P)$ *the inhibition relation,*

$C : F \to 2^L$ *the capacity function,*

is called a Simple Logic Petri Net *(SLPN).*

EXAMPLE 2 (Running example)**.**
Figure 2 shows an SLPN in two states of execution. Places are depicted as circles, transitions as boxes, arcs as arrows. Enabled transitions are grey, black otherwise.

Our main notion is that of a *valid net*. To this end we introduce

DEFINITION 5 (Preset, Postset)**.** *For each $p \in P$ and each $t \in T$ we define the preset and postset of p and t as follows*

$$\begin{array}{rcl} {\bullet p} & = & \{t \in T \mid (t,p) \in F\} \\ p{\bullet} & = & \{t \in T \mid (p,t) \in F\} \\ {\bullet t} & = & \{p \in P \mid (p,t) \in F\} \\ t{\bullet} & = & \{p \in P \mid (t,p) \in F\} \end{array}$$

We assume the following two properties: Firstly, if a variable occurs in the label of an **outgoing** arc from a transition, then this variable must also occur in an **ingoing** arc to this transition. Secondly, we do not allow negative atoms as labels of arcs between transitions and places. Otherwise parts of the net would not be executable or would make no sense at all.

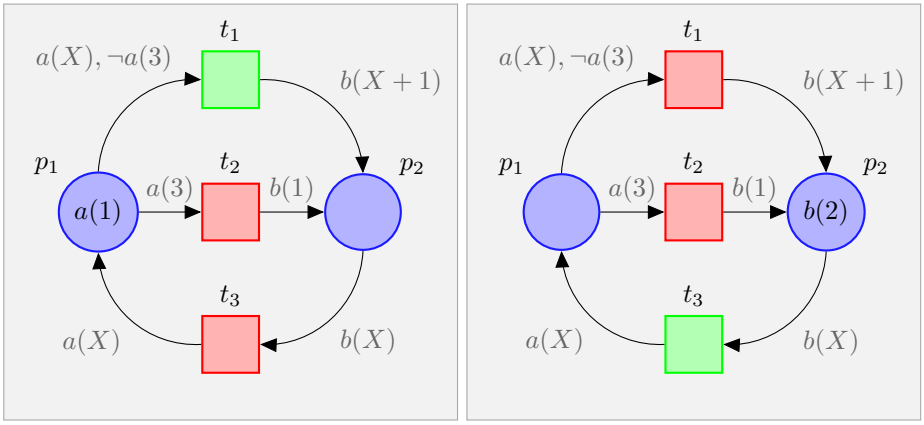

Figure 2: A Simple Logic Petri Net (*SLPN*) $\mathcal{N}$. On the left is the initial state, t_1 is enabled. The right side shows the state after t_1 has fired. Now t_3 is enabled. This *SLPN* is deadlock-free.

DEFINITION 6 (Valid Net). *A SLPN* $\mathcal{N} = \langle P, T, F, C \rangle$ *is* valid *iff the following hold:*

$$\forall t \in T : \ var\text{-}of\left(\bigcup_{p \in t\bullet} C(t,p)\right) \subseteq var\text{-}of\left(\bigcup_{p \in \bullet t} C(p,t)\right) \tag{1}$$

$$\forall (t,p) \in F : \ C(p,t) \in A^+ \tag{2}$$

This means that all variables that are in the labels of arcs between each transition and its postset are also in the labels of the arcs between each transition and its preset. Otherwise we would have uninitialized variables.

Now we have places and transitions and the arcs between them, together with a function which labels each arc with a literal. We would like to have ground atoms inside the places and the ability to move them through the net. Thus we define the state of the net:

DEFINITION 7 (State). *A* state *is a function* $s : P \to 2^{A^+_{grnd}}$.

We denote the *initial state* by s_0. A state s can be written as a vector as $s = \left[s(p_1) \dots s(p_{|P|})\right]^t$ whereas we might leave out the parentheses of the sets $s(p_i)$ iff the representation is clear.

Before describing the transition between states we need the notion of an *enabled transition*. In each state a Petri net might have none, one or several enabled transitions. This holds for our *SLPN*'s as well as for Petri nets in general:

DEFINITION 8 (Enabling of Transitions, Bindings). *A transition* $t \in T$ *is* enabled *if there is* $B \subseteq VAR \times CONST$:

$\forall p \in {\bullet t}\ \forall a(\vec{t}) \in\ C(p,t) \cap A^+ :\ a(\vec{t})_{[B]} \in s(p)$ *and*
$\forall p \in {\bullet t}\ \forall a(\vec{t}) \in\ C(p,t) \cap A^- : \neg a(\vec{t})_{[B]} \notin s(p)$

B is a (possibly empty) set of variable substitutions. We denote wlog by $Subs(t) = \{B_1, \ldots, B_n\}$ with $n \in \mathbb{N}$ the set of all sets of such variable substitutions with respect to the transition t for which the above holds.

This means that a transition t is enabled if (1) all positive literals that are labels of arcs between t and its preset ${\bullet t}$ are unifiable with the literals in the respective places, and (2) that there is no unification for all the negative literals that are labels of arcs between the transition and its preset. Note that we need $\neg a$ in the second formula, because the places never contain negative atoms (closed world assumption).

We are now ready to define transitions of states. *Firing transitions* absorb certain atoms from the places in the preset and put new atoms into the places in the postset using the variable substitutions:

DEFINITION 9 (State Transition)**.**

$$\forall p \in P :\ s'(p) = s(p)\ \setminus\ \left(\bigcup_{t \in p\bullet, tfires} C(p,t)_{[Bind(t)]}\right) \cup\ \left(\bigcup_{t \in \bullet p, tfires} C(t,p)_{[Bind(t)]}\right)$$

Thus each place receives ground literals from each firing transition in its preset and ground literals are absorbed by all the firing transitions in the postset, thus leading to a new state of the whole system.

EXAMPLE 3 (Running example cont'd)**.**
The initial state is $s_0 = [a(1)\ \ \emptyset]^t$ After firing t_1 we have the state $s_1 = [\emptyset\ \ b(2)]^t$.

4 SLPNs and Büchi Automata

Theorem 1 states that LTL formulæ are equivalent to Büchi automata. How can we construct a Büchi automaton from a *SLPN*? We construct the Kripke structure representing the state-space of the *SLPN*.

DEFINITION 10 (Kripke Structure $\mathcal{K}$)**.** *The tuple $\mathcal{K} = \langle S, \Delta, S_0, L\rangle$ with*

- *$S = \{s_1, \ldots, s_n\}$ the set of states,*
- *$S_0 \subseteq S$ the set of initial states,*
- *$\Delta \subseteq S \times S$ the transition-relation with $\forall s \in S\ \exists s' \in S : (s, s') \in \Delta$,*
- *$L : S \to 2^{AP}$ the labeling-function, where $AP = \{a \bullet p \mid a \in A^+_{grnd}, p \in P\}$ is the set of* atomic propositions *over SLPN*

is called a Kripke structure*.*

For each *SLPN* we can generate a respective Kripke structure by exhaustive enumeration of the state-space of the net. The derived Kripke structure is similar to the *reachability graph* of the *SLPN*:

ALGORITHM 1. $\mathbf{KSfromSLPN}(\mathcal{N})$

$S := \{q_1\}$;
$S_0 := \{q_1\}$;
$\Delta := \emptyset$;
$L := \{(q_1, s_0)\}$;
$queue := \{q_1\}$;
while $queue \neq \emptyset$ ***do***
  $q := removeFirst(queue)$;
  $T := getEnabledTransitions(\mathcal{N}, L(q))$;
  for all $S' \subseteq T, S \neq \emptyset$ ***do***
    $s' := fireTransitions(\mathcal{N}, s, T')$;
    if $\exists q' \in S$ *with* $L(q') = s'$ ***then***
      $\Delta := \Delta \cup \{(q, q')\}$;
    else
      $q' = newNode()$;
      $S := S \cup \{q'\}$;
      $\Delta := \Delta \cup \{(q, q')\}$;
      $L := L \cup \{(q', s')\}$;
      $queue := addLast(queue, a')$;
    end if
  end for
end while
return $\langle S, S_0, \Delta, L \rangle$;

LEMMA 1 ($\mathcal{K}_\mathcal{N}$)**.** *Algorithm 1 generates for each SLPN $\mathcal{N}$ a Kripke structure $\mathcal{K}_\mathcal{N}$.*

This algorithm is not guaranteed to terminate.

EXAMPLE 4 (Running example cont'd.)**.**
Figure 3 shows the Kripke structure generated from $\mathcal{N}$ using our algorithm. The structure is as follows:

$$S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$$

$$S_0 = \{s_1\}$$

$$\Delta = \{(s_i, s_j) \mid j \equiv i + 1 \bmod 6\}$$

$$L(s_1) = [a(1)\ \ \emptyset]^t, L(s_2) = [\emptyset\ \ b(2)]^t, L(s_3) = [a(2)\ \ \emptyset]^t,$$
$$L(s_4) = [\emptyset\ \ b(3)]^t, L(s_5) = [a(3)\ \ \emptyset]^t, L(s_6) = [\emptyset\ \ b(1)]^t$$

Kripke structures can be easily transformed into Büchi automata:

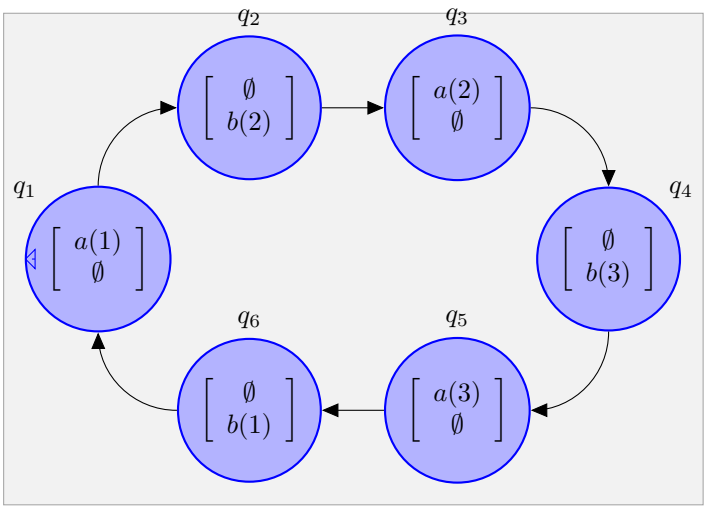

Figure 3: Kripke structure for the *SLPN* in Fig. 2.

THEOREM 2 (Büchi automaton $\mathcal{B}_\mathcal{K}$)**.** *For each Kripke structure $\mathcal{K}$ there exists a Büchi automaton $\mathcal{B}_\mathcal{K}$ that generates exactly those infinite sequences that are equivalent to the possible paths in the Kripke structure.*

sketch. Transforming a Kripke structure into a Büchi automaton is straightforward. Firstly the states and the arcs of the Kripke structure constitute the states and arcs of the Büchi automaton. Secondly introduce the initial state and connect it to the states that were initial states in the Kripke structure. Finally put the labels of the states to the incoming arcs and declare all states *accepting* states. □

EXAMPLE 5 (Running example cont'd)**.**
Figure 4 shows the Büchi automaton $\mathcal{B}_\mathcal{N}$ generated from the SLPN $\mathcal{N}$ in Fig. 2. We are only interested in the truth value of the atomic proposition $b(3) \bullet p_2$ which we denote by ap and $\neg ap$ respectively.

5 Model Checking SLPNs

We reduce model checking to the emptiness check of a Büchi automaton.

THEOREM 3 (Reducing Model Checking to Büchi Automata)**.** *The model checking problem $\mathcal{N} \models \alpha$ for a SLPN $\mathcal{N}$ and an LTL formula α is equivalent to the emptiness problem of the intersection automaton $\mathcal{B}_\mathcal{N} \cap \mathcal{B}_{\neg\alpha}$.*

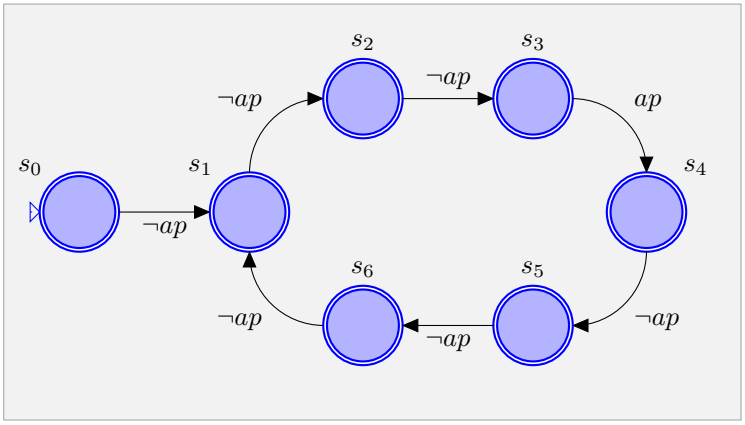

Figure 4: Büchi automaton for the *SLPN* in Fig. 2 with filtered alphabet: the atomic proposition ap represents $b(3) \bullet p_2$. Note that the automaton is filtered: other atoms in p_2 are ignored as well as p_1 is ignored completely.

Proof. Since $\mathcal{B}_{\mathcal{N}}$ generates all possible states of $\mathcal{N}$ (Lemma 1 and Theorem 2) and $\mathcal{B}_{\neg\alpha}$ generates all sequences in which α does not hold, the intersection automaton generates all states in which the formula does not hold. If the generated language is empty α holds in all states. □

LEMMA 2 (Folklore)**.** *Solving the emptiness problem of a Büchi automaton is equivalent to finding a strongly connected component of the automaton-graph that contains at least one initial and one final state.*

Putting our lemmas and theorems together, we get the following

ALGORITHM 2. *Given an LTL-formula α and an SLPN $\mathcal{N}$ the* model checking problem *$\mathcal{N} \models \alpha$ can be solved as follows:*

1. *generate the Büchi-Automaton $\mathcal{B}_{\mathcal{N}}$ from the SLPN by considering the Kripke structure representing the state-space of $\mathcal{N}$*
2. *generate the Büchi-Automaton $\mathcal{B}_{\neg\alpha}$ from the formula α using Gerth's algorithm*
3. *solve the emptiness problem for the Büchi-Automaton $\mathcal{B}_{\mathcal{N}} \cap \mathcal{B}_{\neg\alpha}$ by searching a strongly connected component that contains an initial state and a final state.*

If a strongly connected component of the intersection-automaton is found it can serve as a counterexample. A powerful feature of the above algorithm is that it works *on the fly*: the parts of the automata $\mathcal{B}_{\mathcal{N}}$ and $\mathcal{B}_{\neg\alpha}$ need to be generated *on demand only*. Thus a strongly connected component can be found *without complete generation of the automata by expanding them in parallel*.

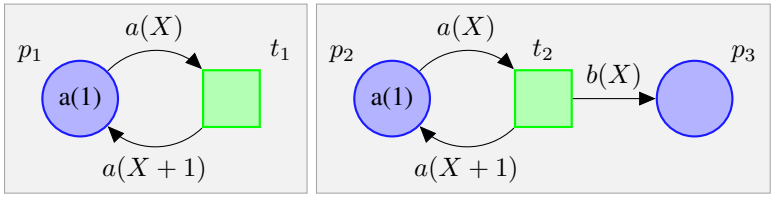

Figure 5: Two *SLPN*'s with infinite Kripke structures. The left side shows how an *SLPN* $\mathcal{N}_1$ that enumerates the natural numbers. The right side shows an even worse scenario: the cardinality of $s(p_3)$ of $\mathcal{N}_2$ increases in each step and is unbounded.

Unfortunately Algorithm 1 has a drawback: some *SLPN*'s have infinite Kripke structures. Figure 5 shows two such *SLPN*'s. This leads to semi-decidability of our model-checking algorithms. Only in the case when a counterexample exists does the algorithm terminate.

6 Related and Future Work

As this is an important area of research, there are many competing approaches. Among the most well-known model checkers, there is SPIN: An efficient verification system for models of distributed software systems [6, 7]. The core is the specification language PROMELA—quite similar to structural programming languages—which is to be used to specify concurrent systems. This specification is translated into an automaton together with the correctness claims expressed in LTL. In contrast to our approach SPIN generates an automaton for each asynchronous process in the system and then generates a new automaton for each possible interleaving of the execution of the processes, whereas we generate the interleavings on the fly. Thus our algorithm might find a counterexample without checking all possible interleavings.

- Recently we have made use of the Petri net infrastructure *Petri Net Kernel* [8] (PNK). It is very useful for developers and scientists, because it allows a powerful architecture for basic Petri net tools like editors and simulators and it also permits the extension by versatile plugins. We use PNK for designing Petri nets and because it comes along with the standardized data exchange format *Petri Net Markup Language* (PNML) based on XML.

- We have implemented an *SLPN*-to-*smodels*-converter which takes an *SLPN* and generates a logic program, whose answer sets represent all possible bounded executions of the net. With it we are able to detect deadlocks [1].

- We are working on an AgentSpeak (F)-to-*SLPN*-converter with which we will test our model-checking-framework.

- We plan to examine the relationship between *SLPN* and the Petri-net-classes P/T nets and Colored Petri nets as well as the operational semantics of *SLPN*.

7 Conclusion

SLPN is a class of specialized Petri nets with logical atoms as tokens and logical literals as labels. It is well suited to describe agent languages based on logical atoms (like the family of AgentSpeak languages).

We gave a short summary on LTL model checking with automata: LTL formulæ as well as Kripke structures can be transformed into Büchi automata, finding a final state that is reachable from an initial state and from itself solves the model checking problem.

Finally we showed how to generate a Büchi automaton from a given *SLPN* and concluded with how to perform LTL model checking on *SLPN*'s.

References

[1] Tristan Behrens and Jürgen Dix. Model checking with logic based petri nets. Technical Report IfI-07-02, Clausthal University of Technology, Dept of Computer Science, May 2007.

[2] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000. ISBN 0262032708.

[3] Rene David and Hassane Alla. *Discrete, Continuous, and Hybrid Petri Nets*. Springer Verlag, 2005. ISBN 3540224807.

[4] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. 1990.

[5] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[6] Gerard J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003. ISBN 0321228626.

[7] Gerard J. Holzmann. Software model checking with spin. *Advances in Computers*, 65:78–109, 2005.

[8] Ekkart Kindler and Michael Weber. The petri net kernel - an infrastructure for building petri net tools. *International Journal on Software Tools for Technology Transfer*, 3(4):486–497, 2001.

[9] Wolfgang Thomas. Automata on infinite objects. In Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 133–164, 1990.

[10] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344. IEEE Computer Society, 1986.